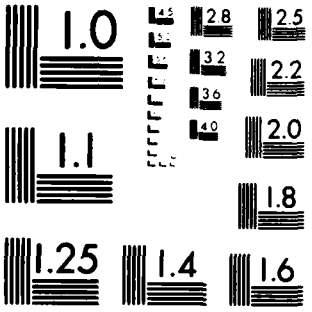
AD-A144 381 ALGORITHMS AND FILE STRUCTURES FOR COMPUTATIONAL GEOMETRY(U) INSTITUT FUER INFORMATIK ZURICH (SWITZERLAND) J NIEVERGELT DEC 82 DAJA37-82-C-0058 1/ 0

UNCLASSIFIED F/G 12/1 NL

END
DATE FILMED
9 84
DTIC



Quarterly Progress Report #3, December 1982

ERO Contract No. DAJA 37-82-C-0058

Algorithms and file structures for computational geometry

Principal Investigator:

Prof. Jurg Nievergelt,
Informatik, ETH,
CH-8092 Zurich, Switzerland

Algorithms and file structures for computational geometry

K. Hinrichs, J. Nievergelt, Institut für Informatik, ETH, CH-8092 Zürich

Abstract

Algorithms for solving geometric problems and file structures for storing large amounts of geometric data are of increasing importance in computer graphics and computer-aided design. As examples of recent progress in computational geometry, we explain plane-sweep algorithms, which solve various topological and geometric problems efficiently; and we present the grid file, an adaptable, symmetric multi-key file structure that provides efficient access to multi-dimensional data along any space dimension.

Introduction

Today there is a great variety of tools to help the application programmer for solving his problems. Basic tools are algorithms and data structures with their corresponding elementary access operations. It is the task of the programmer to represent the objects he is working on by data structures, and to reduce the operations he wants to perform on these objects first to known algorithms, second to elementary operations on data structures, as shown in Fig. 1.

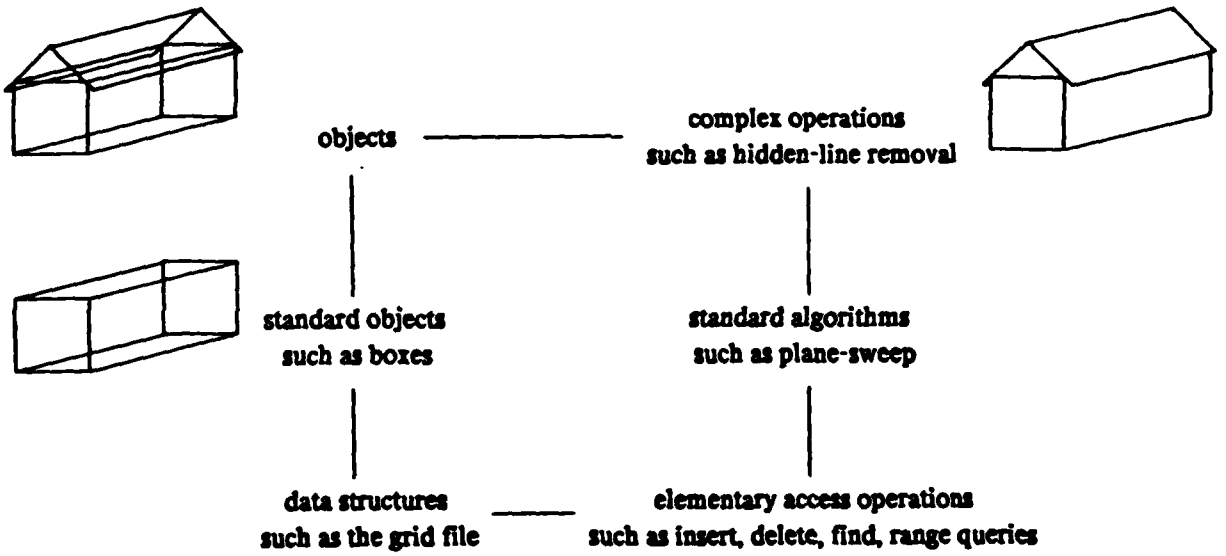


Fig. 1: The role of algorithms and data structures.

We will consider algorithms and file structures for geometric applications. In section 1 plane-sweep algorithms are discussed as an example of a class of algorithms for solving a large number of geometric problems. The grid file, an adaptable, symmetric multi-key file structure is presented in section 2. Section 3 shows how to apply the grid file to storing geometric objects.

Supported in part by the European Research Office, US Army, under contract DAJA37-82-C-00589, R&D 4093-CC.

1. Geometric algorithms

1.1 Application areas for geometric algorithms

One of the objectives of *computational geometry* is to relate the geometric properties of objects to the complexity of algorithms that manipulate them. An important benefit of such a study is the development of efficient algorithms for applications in which geometry plays a major role, such as computer graphics, computer-aided design or cartography.

In computer graphics and computer-aided design hidden-line and hidden-surface elimination problems must often be solved. If a 3-dimensional scene is projected into a plane those parts of an object which are hidden by other objects must be eliminated. In cartography efficient algorithms are needed for the computation of areas or checking the consistency of geographical data bases.

Geometric applications frequently involve finding the intersection of objects. For example a plane polygon is simple if and only if no two of its edges intersect. In VLSI (Very Large Scale Integration) design all intersecting pairs among a set of rectangles in a plane must be computed for design rule checking. We will consider the problem of finding all intersections among a set of intervals on the real line and the problem of finding all intersections among a set of line segments in the plane.

The importance of efficient algorithms for these problems is becoming more and more apparent as applications increase rapidly. For instance a single VLSI-chip or a complicated scene for graphic display may contain a hundred thousand components, and data bases for geographical data may consist of millions of data. For these problems even algorithms with a quadratic asymptotic time complexity are impractical.

1.2 Interval intersection

Suppose we are given n intervals on the real line and are asked to *report all intersecting pairs of intervals* (Fig. 2).

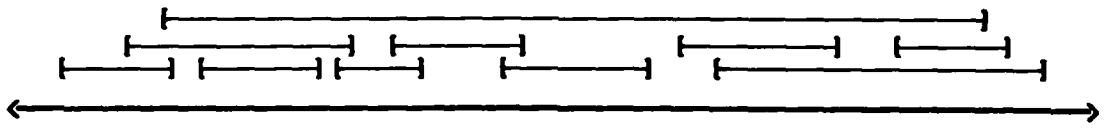


Fig. 2: Interval intersection

Since there are $(n^2 - n)/2$ pairs the asymptotic time complexity of a brute force algorithm is $O(n^2)$. This time bound can be improved by making use of the fact that there is a natural ordering relation on intervals. If A and B are given intervals then either A is to the left of B, A is to the right of B, or they intersect. For simplicity of representation we assume that the intervals have no endpoints in common. First we sort the 2n endpoints. Consider an initially empty data structure Q for the storage of intervals with operations *insert*, *delete* and *report all intervals in Q*. Now the 2n endpoints are scanned from left to right. At each moment Q will contain those intervals of which the left endpoint, but not the right, has been encountered. When reaching the left endpoint of an interval, all intervals which are currently stored in Q are reported as intersecting the interval, which will be inserted as next step. An interval will be deleted from Q when its right endpoint is encountered. It is obvious that all pairs of intersecting intervals are reported. Sorting the endpoints costs O(n log n) time [AHU 74]. The data structure Q is a *priority queue* which can be implemented by a *heap* or a *balanced binary tree* [AHU 74]. The key under which each interval will be stored is its right endpoint. Insertion and deletion of one element in such a data structure cost O(log k) time, if k is the number of elements in this data structure. O(log n) is therefore in our case an upper bound for the cost of each of these two operations. Since we scan through 2n endpoints the whole scanning costs O(n log n) time. The cost of reporting one intersecting pair is O(1). If there are s intersecting pairs the total amount of time for reporting them will be O(s). Altogether we get an asymptotic time complexity of O(n log n + s).

1.3 A plane-sweep algorithm for line segment intersection

Let us consider the following problem: given n line segments in the plane, *report all intersecting pairs of line segments.* This can also be solved in a brute force way in $O(n^2)$ time by inspecting all pairs of line segments. The algorithm given above for the interval intersection problem suggests that there is a similar algorithm for the 2-dimensional case which runs in time $O(n \log n + s)$, where s is the number of intersecting pairs of line segments. Because there is no geometrically induced ordering relation on line segments in the plane which is related in a natural way to the intersection of line segments the above algorithm can not be used to solve this problem. We have to define a new ordering relation. To simplify the representation we assume that no segment is vertical and that no three segments meet in a single point. Also, all of the endpoints of the line segments are assumed to have different x-coordinates. These assumptions do not change the asymptotic running times of the algorithm which will be described.

Consider a vertical line S sweeping from left to right across the plane (Fig. 3).

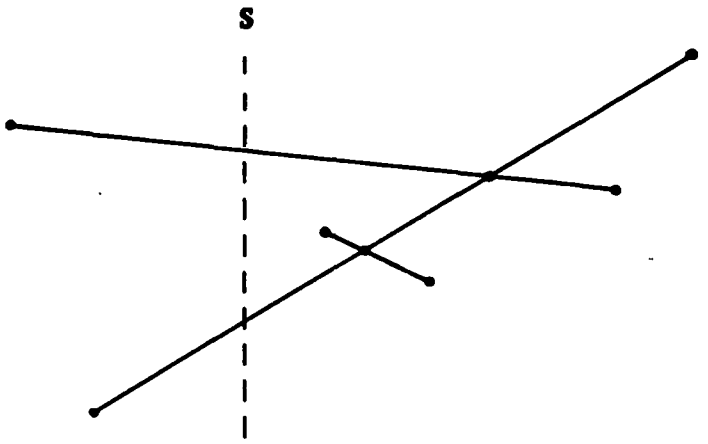


Fig. 3: S sweeps from left to right.

This vertical line defines a total ordering on the set of line segments which are currently cut by it. Let A and B be two line segments which are both cut by S at position x. A is defined to be above B with respect to x if the intersection point of A with S is above the intersection point of B with S. Now it is important to realize that if segments A and B intersect, then there is some x for which A and B are consecutive in the total ordering of the set of line segments which are cut by S at position x. To detect all intersections it is necessary to maintain this total ordering as S sweeps the plane. The total ordering changes only if S encounters a left or right endpoint of a line segment or an intersection point of two line segments. These points will therefore be called transition points. In the case of a left endpoint the corresponding line segment has to be inserted into the total ordering, and then checked to see whether it intersects its upper or lower neighbour. In the case of a right endpoint the corresponding line segment has to be deleted from the total ordering, and the two line segments which become consecutive in the total ordering have to be checked to see whether they intersect. When S meets the intersection of two line segments they are exchanged in the total ordering, and then the two new pairs of neighbours in the total ordering have to be checked for intersection.

Now the algorithm proceeds in a manner similar to the 1-dimensional algorithm given earlier. It operates on two data structures, the *X-queue* and the *Y-table*, which are common to all plane-sweep algorithms. As the vertical line S that sweeps the plane advances in the direction of the x-axis the X-queue contains all the left and right endpoints of line segments and all the intersection points discovered so far which lie on the right side of S and therefore have not yet been processed. These points are sorted according to their x-value and are assigned a type depending on whether they are a left or right endpoint or an intersection point. The X-queue is a *priority queue* that supports the following operations within time bound $O(\log k)$ when it contains k entries:

- MIN: find and remove the entry with minimal x-coordinate.
- INSERT: insert a new entry with a given x-coordinate.

Heaps or *balanced trees* are suitable for implementing priority queues [AHU 74]. The initial contents of the X-queue are the 2n endpoints of the given line segments, sorted by their x-values; at the end the X-queue will be empty. At each transition, the point which defines this transition will be removed, and at most two intersection points are inserted into X. During execution a total of $2n+s$ points, where s is the number of intersecting pairs of line segments, move through the X-queue; the maximal number of entries at any time is therefore less than $2n+s$. Since $s = O(n^2)$, any operation on the X-queue can be done in time $O(\log(n+s)) = O(\log n)$.

The Y-table contains all the information about the total ordering on the set of line segments which are currently cut by the vertical line S. It has an entry for each such line segment, containing a formula that defines the segment, so that for any x the corresponding value $y = ax+b$ can be obtained in time O(1). At the beginning and at the end Y is empty. Y is a *dictionary* that supports operations FIND, INSERT, DELETE, PREDECESSOR and SUCCESSOR within time bound O(log k) when it contains k entries [AHU 74]. We can tailor the exact definition of these operations to the specific use we will make of them, thus postulating dictionary operations that are easily derived from the standard ones:

- FIND(P): given a point P = (x, y, t), obtain one of the following results depending on the type t of P:
  right end point: the unique line segment s whose right end point is P;
  intersection point: the two line segments whose intersection point is P.
- INSERT(s): given a line segment s, insert s at the proper place determined by the y-values of s and the line segments stored in the Y-table at the current x-value.
- DELETE(s): given a line segment s, delete s.
- SUCCESSOR(s): given a line segment s and the current x-value, return the neighboring line segment just above s.
- PREDECESSOR(s): given a line segment s and the current x-value, return the neighboring line segment just below s.

A dictionary with k entries can be implemented so as to support the above operations within time O(log k) by any of several types of balanced trees [AHU 74]. Binary search for a given y-value is performed by evaluating the linear formulas $y = ax + b$ stored as segment entries along a root-to-leaf path. Since there are never more than n entries in the Y-table, any of the above operations can be done in time O(log n).

The algorithm that sweeps the plane and reports the pairs of intersecting line segments has the following simple overall structure:

```
procedure SWEEP;

  X ← 2n left and right endpoints of given line segments, sorted by x-coordinate
  Y ← empty
  while X # empty do
  begin
    P ← MIN(X)
    TRANSITION(P)
  end

end of SWEEP;
```

Procedure TRANSITION is the advancing mechanism of SWEEP, and encompasses all the work involved in processing one point P and moving the vertical sweep line to the next transition point; in this process it updates the corresponding data structures and builds up the result in an output structure. TRANSITION is invoked exactly $2n+s$ times, if s is the number of pairs of intersecting line segments. We will show that one invocation uses O(log n) time, and thus establish an $O((n+s)\log n)$ time bound on the performance of SWEEP.

In TRANSITION a function INTERSECT(s,t) checks in time O(1) whether two line segments intersect and if so, inserts the intersection point into X in time O(log n) as we have seen above. Permuting two line segments s and t does not alter the structure of Y and can be done in time O(1).

procedure TRANSITION(P) breaks into three cases depending on the type of P:

```
case left end point:
  s ← line segment starting at P
  INSERT(s)
  h ← SUCCESSOR(s)
  l ← PREDECESSOR(s)
  INTERSECT(s,h)
  INTERSECT(s,l)
end of case left end point;

case right end point:
  FIND(P) yields the unique line segment whose right endpoint is P
  h ← SUCCESSOR(s)
  l ← PREDECESSOR(s)
  INTERSECT(h,l)
  DELETE(s)
end of case right end point;

case intersection point:
  FIND(P) yields the two line segments s and t whose intersection point is P
  (suppose that s = SUCCESSOR(t))
  h ← SUCCESSOR(s)
  l ← PREDECESSOR(t)
  INTERSECT(h,t)
  INTERSECT(l,s)
  permute s and t
end of case intersection point;

end of TRANSITION;
```

All three cases of procedure TRANSITION are built from the same building blocks in slightly different combinations. The operations performed can all be done in time O(log n) as we have seen above. Therefore one invocation of TRANSITION uses O(log n) time. Since the algorithm that sweeps the plane makes $2n+s$ transitions, it runs in time $O((n+s)\log n)$ as stated earlier. Note that if s is very close to n^2 then the running time of the algorithm is actually greater than the $O(n^2)$ time of the brute force algorithm. But in most real applications s is bounded by O(n).

It's still an open question whether there exists an algorithm which finds the s intersecting pairs among a set of n line segments in the plane in time $O(n \log n + s)$. Shamos and Hoey [ShHo 76] showed that this time complexity is a lower bound for the problem.

1.4 Common aspects and generality of plane-sweep algorithms

If presented in a sufficiently abstract form, most *plane-sweep algorithms* discussed in the literature can be cast into the same mold. It is useful to understand them as a general class of algorithms that can solve many geometric problems in time $O(n \log n)$ instead of the $O(n^2)$ required by naive algorithms.

Many plane-sweep algorithms use three data structures to keep track of the work still to be done and of the results already accumulated; these are independent of the type of objects being processed. In the most general case these are:

The X-queue:	a *priority queue* that supports the operations MIN and INSERT in time O(log n).
The Y-table:	a *dictionary* that supports FIND, INSERT, DELETE, PREDECESSOR and SUCCESSOR in time O(log n).
The R-structure:	a *list structure* on which APPEND and CONCATENATE operations are performed in time O(1).

We don't need the R-structure for the line intersection problem.

The generality of plane-sweep algorithms can be assessed by considering the following questions, which can all be answered within the asymptotic time O((n+s)log n). It includes most questions of practical importance in graphic applications. It may come as a surprise that all these properties can be determined merely by accumulating local information at the moving front.

Topological problems:
- Construct an adjacency graph: each region is represented by a node, an edge represents a pair of adjacent regions.
- Construct a region enclosure tree: the sons of each node represent the regions directly surrounded by the region of the father node.

Geometric problems:
- Compute the area of each region.
- Determine the maximal or minimal width (in the y-direction) of each region; check whether certain minimal distances between regions are maintained.
- List the boundary of each region in cyclic order, and compute its length.

The efficacy of plane-sweep algorithms is based on the transformation of a 2-dimensional problem into a sequence of 1-dimensional problems. The 1-dimensional problem turns out to be significantly simpler than the original 2-dimensional one for the following reason: geometric objects placed in 2-dimensional space can rarely be totally ordered in a useful way. In the absence of a total order, efficient logarithmic search techniques, such as binary search, are inapplicable; linear search must be used, leading to $O(n^2)$ instead of $O(n \log n)$ algorithms. On the other hand, the projections onto the 1-dimensional scan line of the line segments that define these 2-dimensional objects can be totally ordered (by y-coordinate), permitting logarithmic access time to any object intersected by the scan line. This leads to O(n log n) algorithms.

More on plane-sweep algorithms can be found in [ShHo 76], [BeOt 79], [BeWo 80], [Bro 81], [McCr 82] and [NiPr 82].

2. The grid file

The grid file emerged as an answer to the various deficiencies encountered when traditional file structures are used for multi-key access to dynamic files [NHS 81]. It is a dynamic multi-key file structure that adapts gracefully to its contents under insertions and deletions, and achieves an upper bound of two disk accesses for single record retrieval; it also handles range queries and partially specified queries efficiently and preserves the order defined on each attribute domain in such a way that records which are near in the domain of any attribute are likely to be in the same physical storage block.

The starting point is the extreme solution given by the *bitmap representation* of the attribute space, which reserves one bit for each possible record in the space, whether it is present in the file or not. In a k-dimensional bitmap the combinations of all possible values of k attributes are represented by a bit position in a k-dimensional matrix (Fig. 4). A 1 indicates the presence of a record with attribute values determined by its position in the map, a 0 indicates absence.

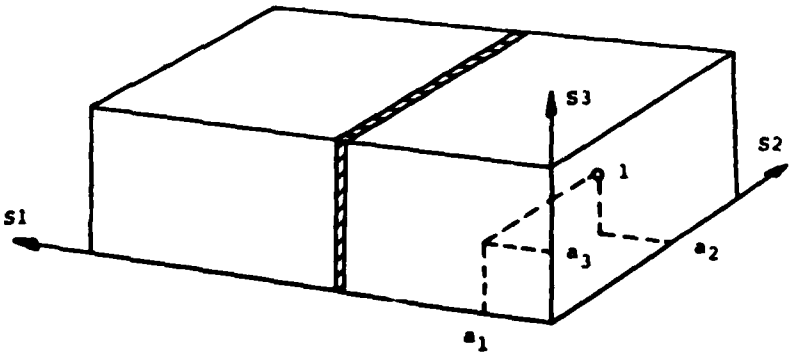


Fig. 4: A 3-dimensional bitmap.

FIND reduces to direct access, INSERT / DELETE requires that a position in the bitmap be set to 1 or 0 respectively, and NEXT in any dimension requires a scan until the next 1 is found. For realistic applications, this bitmap is impossibly large and has to be compressed. In maintaining a dynamic partitioning (directory) on the space of all key-values one approximates the bitmap through compression. Assuming independent attributes, but not necessarily uniform distributions, the embedding space from which the data is drawn is partitioned in a *grid-like* fashion: each region boundary cuts the entire search space in two. All attributes are given the same priority when being partitioned.

The following terminology and notation is used in the example of the 3-dimensional case. On the record space $S = X \times Y \times Z$, a grid partition $P = U \times V \times W$ is obtained by imposing intervals on each axis and dividing the record space into blocks, called grid blocks, as shown in Fig. 5. The picture also shows the effect of refining P by splitting interval v_1.

Record space: $S = X \times Y \times Z$
Grid Partition: $P = U \times V \times W$
Intervals of the partition: $U = (u_0, u_1, \ldots, u_k)$
$V = (v_0, v_1, \ldots, v_l)$
$W = (w_0, w_1, \ldots, w_m)$

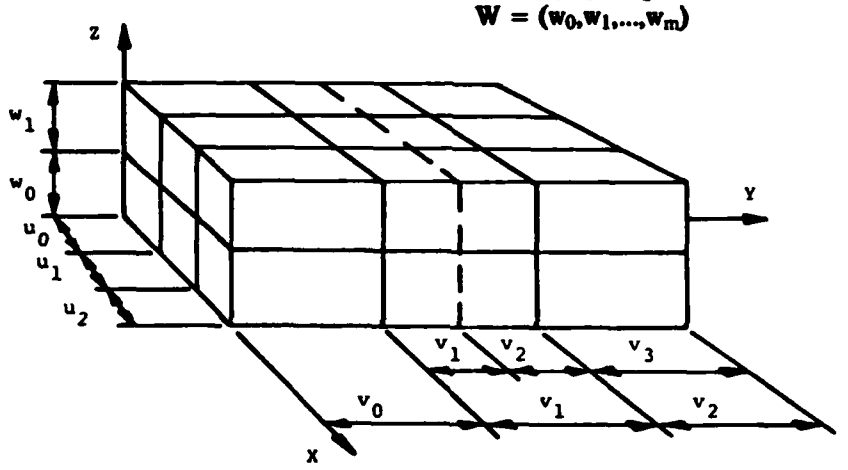


Fig. 5: A 3-dimensional record space $X \times Y \times Z$, with a grid partition $P = U \times V \times W$.

During operation of a file system the underlying partition of the search space needs to be modified in response to insertions and deletions. The grid partition $P = U \times V \times W$ is modified by altering only one of its components at a time. A 1-dimensional partition is modified either by *splitting* one of its intervals in two, or by *merging* two adjacent intervals into one. Fig. 5 shows this for the partition V.

In order to obtain a file system, operations that relate grid blocks and records to each other are needed, such as: find the grid block in which a given record lies, or list all records in a given grid block. The data structure used to organize records within a bucket is of minor importance for the file system as a whole; the structure used to organize the set of buckets, on the other hand, is the heart of a file system. The set of buckets of a given file system is usually managed through a directory. The purpose of the grid directory is to maintain the dynamic correspondence between grid blocks in the record space and data buckets. For reasons of access efficiency, all records in one grid block must be stored in the same bucket. To avoid low bucket occupancies several grid blocks may share a bucket (Fig. 6). Such a set of grid blocks is called a *bucket region*. Bucket regions are only allowed to have the shape of a k-dimensional rectangular box. These convex regions of buckets are pairwise disjoint, together they span the space of records.

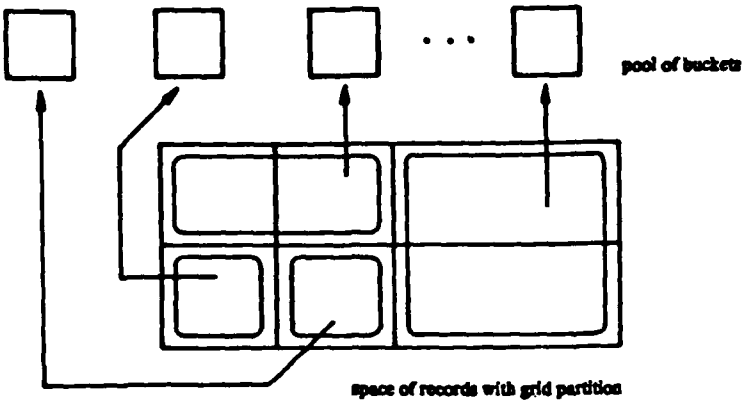


Fig. 6: A convex assignment of grid blocks to buckets.

A grid directory consists of two parts:
- a dynamic k-dimensional array called the grid array; its elements (pointers to data buckets) are in 1:1 correspondence with the grid blocks of the partition;
- k 1-dimensional arrays called linear scales; each scale defines a partition of a domain S.

The grid array is likely to be large and must be kept on disk, but the linear scales are small and can be kept in central memory.

The following example illustrates how the grid file is accessed. Consider a record space with attribute *year* with domain 0 .. 2000, and attribute *initial* with domain a .. z. Assume that the distribution of records in the record space is such as to have caused the following grid partition to emerge:

year = (0, 1000, 1500, 1750, 1875, 2000); initial = (a, f, k, p, z).

A `FIND` for a fully specified query, such as `FIND[1980,w]`, is executed as shown in Fig. 7. The attribute value 1980 is converted into interval index 5 through a search in scale *year*, and w is converted into the interval index 4 in scale *initial*. The interval indices, 5 and 4, provide direct access to the correct element of the grid directory, where the bucket address is located.

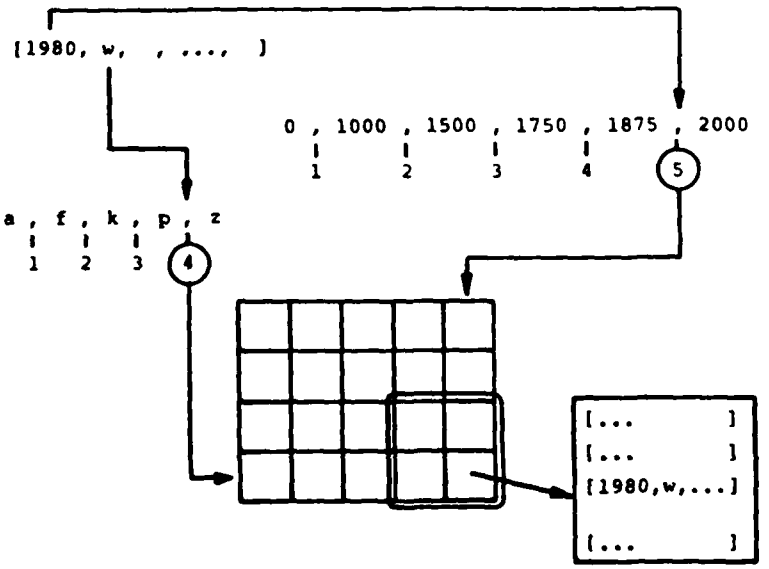


Fig. 7: Retrieval of a single record in the grid file.

More details on the grid file can be found in [NHS 81].

3. Geometric data base operations on a grid file

3.1 The scope of data structures in computational geometry

The problems that arise in *computational geometry* can be divided into two classes depending on the applicability of data structures or algorithms (Fig. 8). Of primary interest for data structures are problems which involve only simple objects and simple operations. If objects or operations get more complex, tools become algorithmic in type, for instance plane-sweep algorithms.

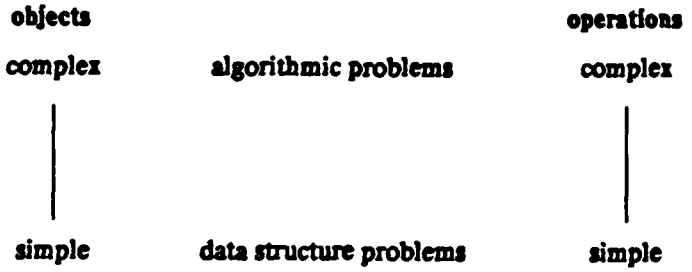


Fig. 8: Scope of data structures in computational geometry

Sometimes it is possible to reduce problems that are determined by complex objects or operations to data structure problems by replacing complex by simple objects or decomposing complex into simple operations. For instance if the intersecting pairs among a set of complex objects have to be determined, in first approximation all the objects may be replaced by containers which are of a simple type, and these are checked for intersection (Fig. 9). Then only those objects whose containers intersect have to be checked for intersection.

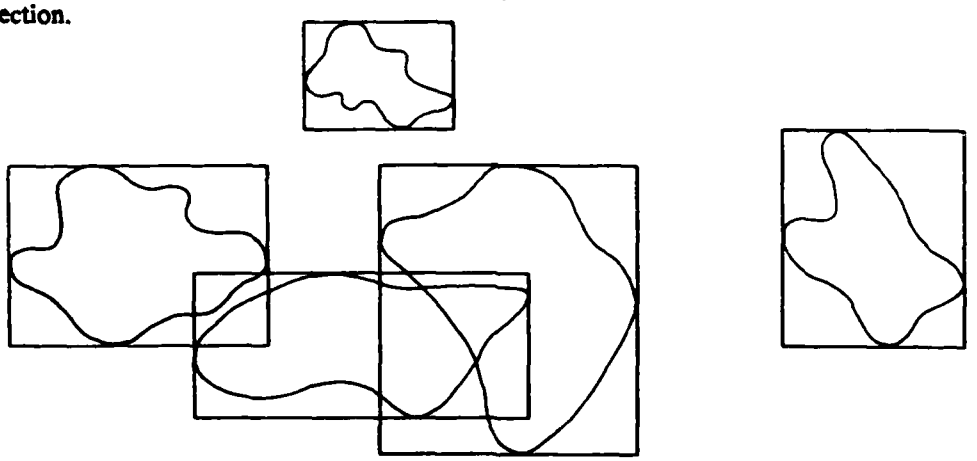


Fig. 9: Replacement of complex objects by containers.

3.2 Simple objects and corresponding operations

Examples of the basic geometric objects under consideration are points, line segments, rectangles, circles, triangles, k-dimensional rectangular boxes and k-dimensional spheres.

Examples of simple operations or queries are:
- identity query: find a special object;
- point in object: find all objects that contain a given point;
- object intersection: find all objects intersecting a given object;
- object containment: find all objects contained in a given object;
- nearest neighbour: find the nearest neighbour to a given object;
- range query: find all objects in a special range.

3.3 Representation of simple objects as points in higher-dimensional spaces

Many simple geometric objects are defined by a small, fixed number of parameters. Any object with k parameters can be represented as a point in k-dimensional space. An interval on a straight line may be described by its left and right end points, x_l and x_r (Fig. 10a). *Its representation in 2-dimensional space is* the point (x_l, x_r). No point will lie below the diagonal because we have $x_l <= x_r$. Since in most applications an interval will have a length that is small compared to the length of the range in which the segments lie, all points will be contained in a narrow band parallel to the diagonal (Fig. 10b). For data or file structures that organize the embedding space rather than the data itself, this clustering leads to inefficiencies. If we take the midpoint x_m and the half length d of the segment we get another representation (Fig. 10c). Now it is possible for points to be below the diagonal, and furthermore they will no longer be clustered in a diagonal band. This simple example shows that one has to be careful when choosing parameters for describing geometric objects: the parameters should be independent of each other as much as possible.

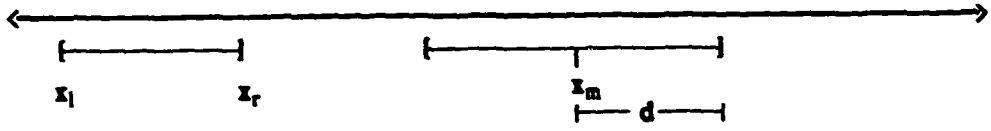


Fig. 10a: Intervals on a straight line.

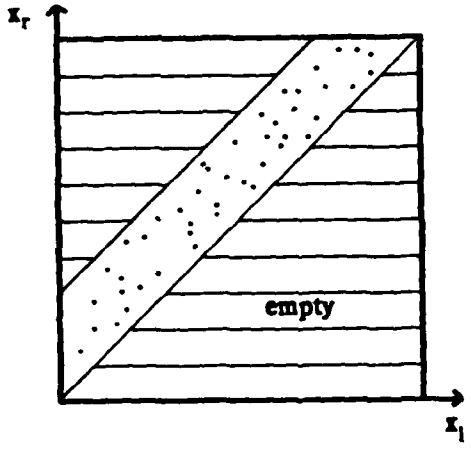


Fig. 10b: Representation of intervals by left and right endpoints.

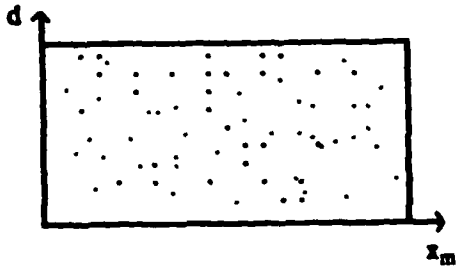


Fig. 10c: Representation of intervals by midpoints and half length.

A circle is given by the centre (x, y) and the radius r and is represented in 3-dimensional space by the point (x, y, r).

An aligned rectangle, i. e. a rectangle with sides parallel to the axes, can be described by its centre (x, y) and half the length of each side, dx and dy; it is represented in 4-dimensional space by the point (x, y, dx, dy).

Line segments in the plane are represented as points in 4-dimensional space, rectangles as points in 5-dimensional and triangles as points in 6-dimensional space.

3.4 Object intersection and the search region

The example of object intersection will show how to treat basic queries in a set of simple objects if these objects are represented as points in higher-dimensional space. Given a geometric object q we can describe exactly the region in the higher-dimensional space that contains all points representing objects which intersect q.

For instance an interval on a straight line given by its left and right end points x_l and x_r intersects a query interval q with end points q_l and q_r if and only if the inequations $x_l <= q_r$ and $x_r >= q_l$ are satisfied. All intervals which intersect q (Fig. 11a) are represented in 2-dimensional space by points which lie in the region shown in Fig. 11b. If an interval is given by its midpoint x_m and its half length dx, it intersects a query interval q with midpoint q_m and half length dq if and only if the inequations $x_m - dx <= q_m + dq$ and $x_m + dx >= q_m - dq$ are satisfied. In this case all intervals which intersect q are represented in 2-dimensional space by points which lie in the region shown in Fig. 11c.

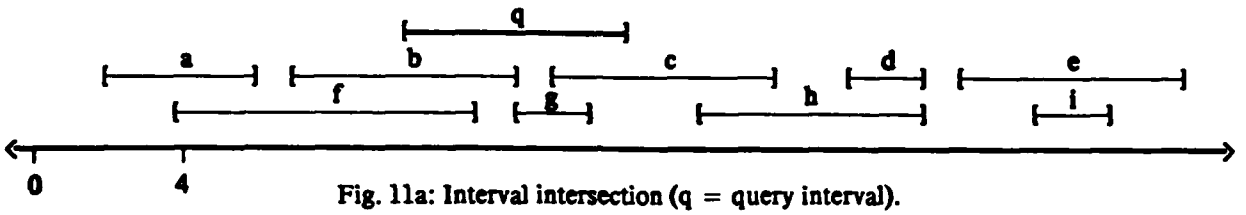


Fig. 11a: Interval intersection (q = query interval).

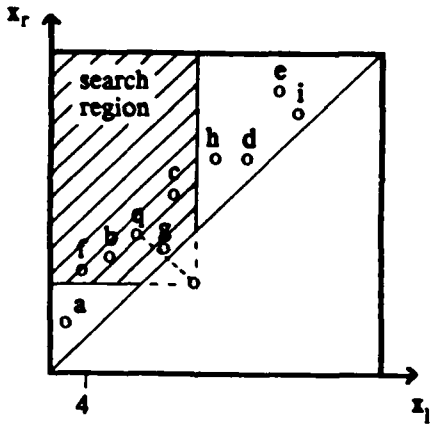


Fig. 11b: Search region for interval intersection (intervals given by left and right endpoints).

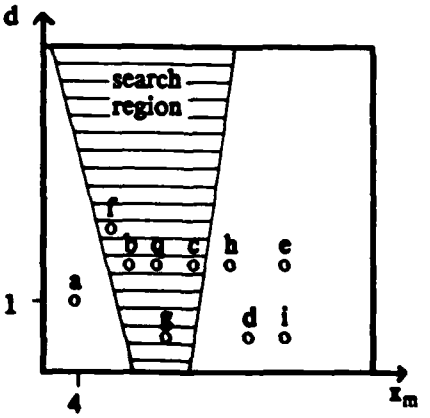


Fig. 11c: Search region for interval intersection (intervals given by midpoints and half length).

All circles which intersect a given circle q with midpoint (X, Y) and radius R (Fig. 12a) are represented by points in 3-dimensional space which lie in a truncated cone (Fig. 12b). The truncated cone is limited in the x-y-plane by the query circle q.

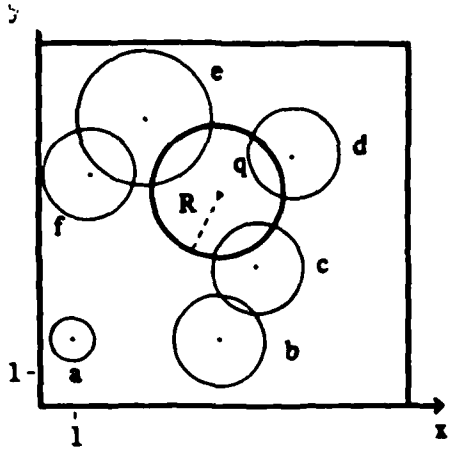


Fig. 12a: Circle intersection (q = query circle).

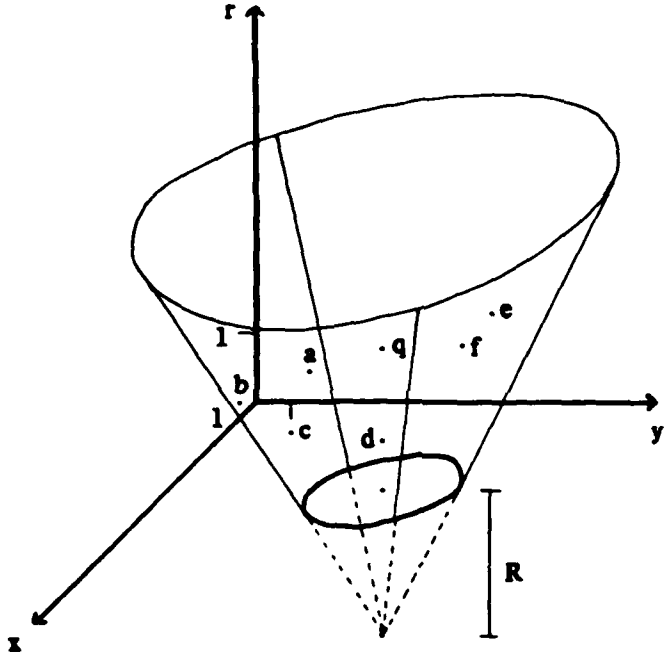


Fig. 12b: Search cone for circle intersection.

In the case of aligned rectangles (Fig. 13a) we get a region in 4-dimensional space which contains all points representing rectangles that intersect the rectangle q given by its centre (X, Y) and half the length of each side, DX and DY (Fig. 13b). The region is shown by its projections in the x-dx-plane and in the y-dy-plane.

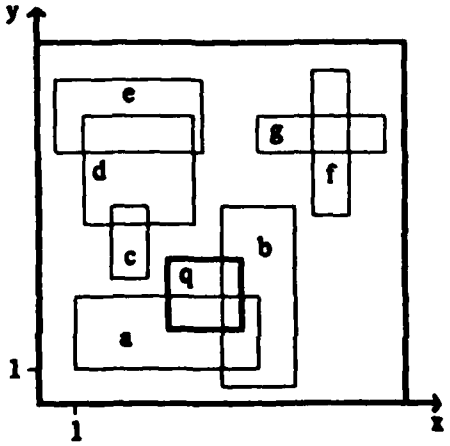


Fig. 13a: Aligned rectangle intersection (q = query rectangle).

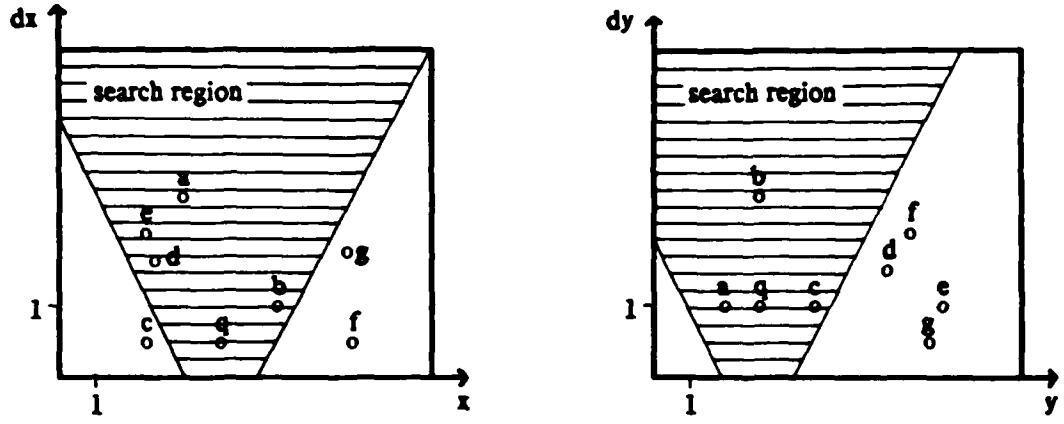


Fig. 13b: Search region for rectangle intersection.

3.5 Space partitioning by file structures

The efficiency of a file structure for storing simple geometric objects depends on how fast the basic queries mentioned in section 3.2 can be answered. Like in the case of object intersection each of these queries defines a coherent search region in the space in which the objects are represented as points. Hence an efficient file structure should preserve *locality:* objects which are represented by points that are near to each other should have a high probability of being stored in the same bucket. Therefore in geometric applications the partitioning of the space by a file structure has a great influence on the efficiency.

The *inverted file* [Knu 73], for example, partitions the 3-dimensional space into slices parallel to the x-y-plane if the r-coordinate is taken as the primary key (Fig. 14). Observing that in the case of circle intersection the search region was a truncated cone (Fig. 12b), it is clear that the inverted file is an unsuitable structure for our problem because the cone is badly approximated by such slices.

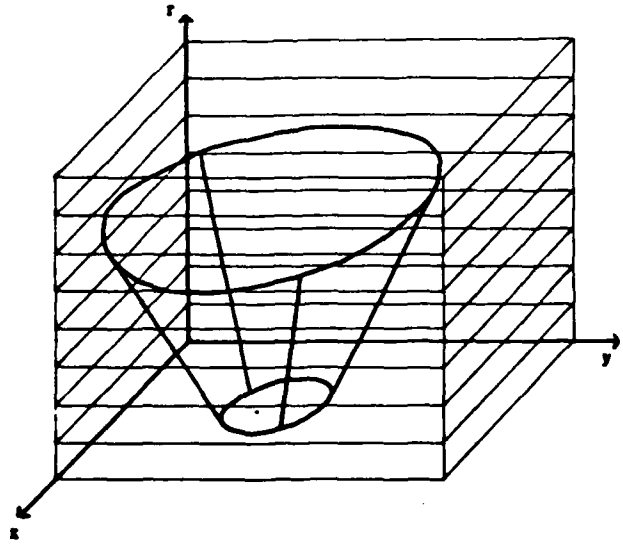


Fig. 14: Space partitioning by an inverted file.

If the file is organized as a *multi-dimensional tree* [Ben 75], [Ben 79] the situation looks much better. Let us consider the 2-dimensional space as an example (Fig. 15). The nodes of the tree contain the coordinates at which the space block that corresponds to this node is split. The leaves of the tree are pointers to buckets containing all the points that lie in the space block which is described by the path from the root to the leaf and called the bucket region. This can be generalized to higher-dimensional spaces. So multi-dimensional trees partition the space into rectangular boxes.

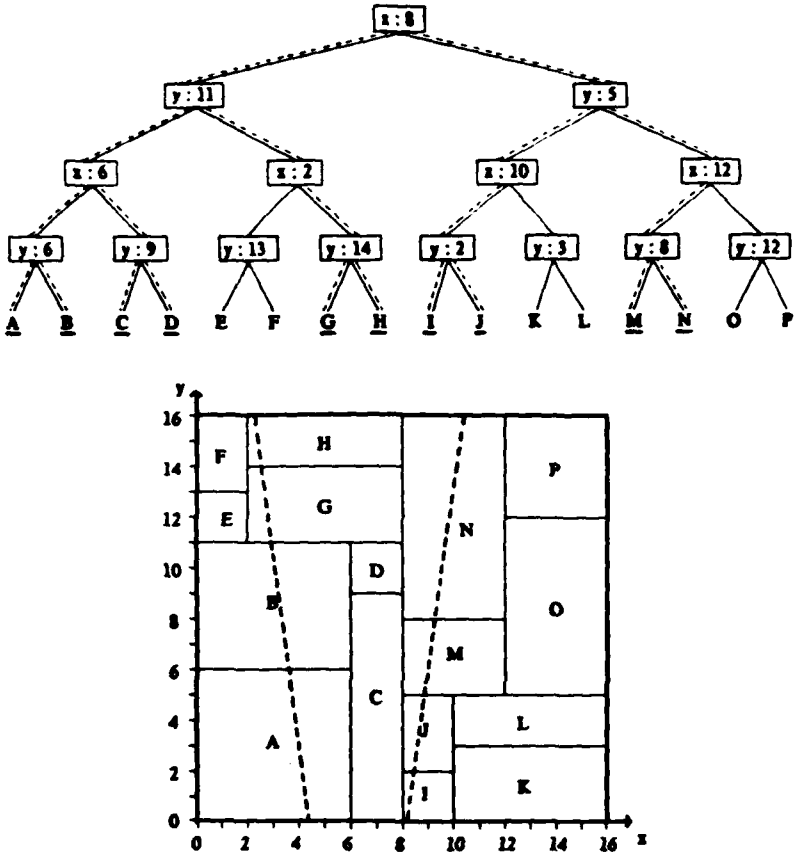


Fig. 15: Space partitioning by a multidimensional tree.

The *grid file* partitions the space in a similar way, with the only difference that each region boundary cuts in two the whole space, and not only one space block (Fig. 16). In the k-dimensional case this grid partition is described by the k scales which are 1-dimensional arrays and define the partitions of the axes, and by the grid directory which is a k-dimensional array. To every space block (grid block) there corresponds an element of the grid directory which is a pointer to the bucket that contains all the points lying in this space block. A bucket may correspond to several space blocks, but the union of all space blocks corresponding to one bucket (the bucket region) is only allowed to have the shape of a k-dimensional rectangular box.

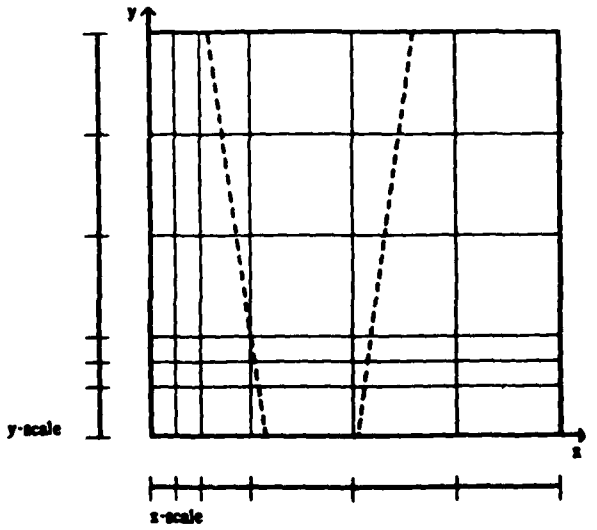


Fig. 16: Space partitioning by a grid file.

In both cases it is clear how to solve the search problem. All buckets whose bucket regions intersect the search region must be computed, because these buckets are exactly those which could contain objects to be searched for.

From the point of view of space partition a file organized by a multi-dimensional tree is equivalent to a grid file. Both partition the space into *near-cubes* and therefore preserve locality in the above sense. The difference between the two comes up in computing the pointers to the relevant buckets, as will be seen in the next section.

3.6 Effective use of central memory to reduce the number of disk accesses

In the last section we stated that file structures for geometric applications should preserve locality. For a fast answer to a basic query it is also necessary that those space blocks which intersect the search region are computable in an efficient way. Since the multi-dimensional tree and the grid directory are normally stored on disk, it should be possible to get the pointers to buckets that correspond to contiguous space blocks in as few disk accesses as possible. Therefore these structures should also preserve *contiguity of space blocks*, i. e. pointers to buckets that correspond to contiguous space blocks should have a high probability of being stored in the same bucket.

Trees are commonly implemented as list structures. One has to follow pointer chains to compute the space blocks and to get the pointers to the corresponding data buckets. Fig. 17 shows how a tree is distributed among buckets, which are denoted by dashed lines.

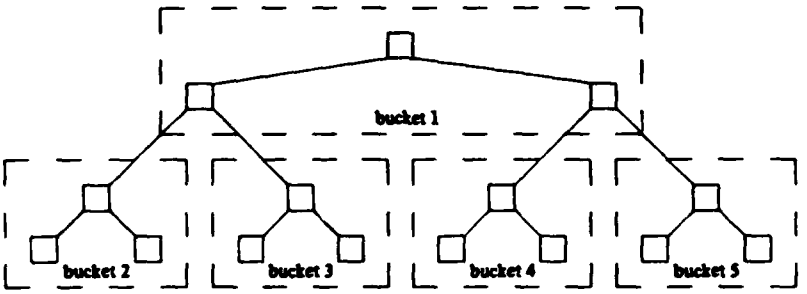


Fig. 17: Paginating a tree.

Since one chain may be distributed among several buckets and contiguous space blocks may correspond to non-contiguous subtrees (Fig. 15), it is clear that often more than one disk access is necessary to compute a space block and to get the pointer to the corresponding data bucket, or to get the pointer to a data bucket that corresponds to a contiguous space block from a given one. In queries we have to follow several pointer chains to determine the space blocks that intersect the search region. We can't compute these space blocks without access to the tree. The dashed lines in Fig. 16 mark the paths which have to be followed to get the pointers to those data buckets that correspond to space blocks which intersect the search region.

Applying the grid file the situation improves significantly. Those space blocks that intersect the search region can be computed without any disk accesses with the aid of the scales which are kept in central memory. In the concept of *resident grid directory* [NHS 81], which manages the grid directory on disk and is also kept in central memory like the scales, the space blocks are distributed among buckets as shown in Fig. 18. Since contiguous space blocks correspond to contiguous grid directory elements which are likely to be stored in the same bucket, only few disk accesses are necessary to examine all the grid directory elements which contain pointers to the relevant buckets.

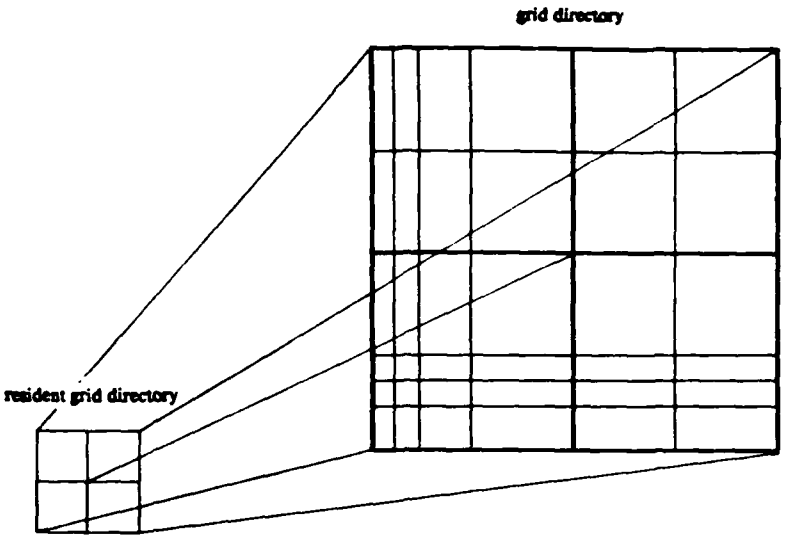


Fig. 18: Distribution of grid directory among buckets.

4. Implementation of the grid file for storing simple geometric objects

A first version of the geometric grid file has been implemented on the *Lilith* personal computer which has been developed at ETH Zürich. The programming language is MODULA-2. In this version the grid directory is kept in central memory. There are INSERT, DELETE and FIND operations for single records. Since we are interested in the application of the grid file for storing simple geometric objects a graphical user interface has been written for testing and demonstrating the grid file. This program allows to insert, delete and search aligned rectangles, which are represented as points in 4-dimensional space and displayed on the screen. In a next version of the grid file program the resident grid directory will be implemented, which is a kind of small grid directory on the proper grid directory and suits well for storing geometric objects. It will also be possible to do range queries. This will then be applied to the demonstration program for doing geometric queries like intersection or containment.

References

[AHU 74] A. Aho, J. E. Hopcroft, J. D. Ullmann: The Design and Analysis of Computer Algorithms, Addison Wesley, 1974.

[Ben 75] J. L. Bentley: Multi-dimensional Search Trees used for Associative Searching, CACM 18, 9, 1975, 509 - 517.

[Ben 79] J. L. Bentley: Multi-dimensional Binary Search Trees in Database-Applications, IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, 1979, 333 - 340.

[BeOt 79] J. L. Bentley, T. A. Ottmann: Algorithms for Reporting and Counting Geometric Intersections, IEEE Transactions on Computers, Vol C-28, No. 9, 1979, 643 -647.

[BeWo 80] J. L. Bentley, D. Wood: An optimal worst case algorithm for reporting intersections of rectangles, IEEE Transactions on Computers, Vol C-29, No. 7, 1980, 571 - 576.

[Bro 81] K. Q. Brown: Comments on "Algorithms for Reporting and Counting Geometric Intersections", IEEE Transactions on Computers, Vol C-30, No. 2, 1981, 147 - 148.

[Knu 73] D. E. Knuth: The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison Wesley, 1973.

[McCr 82] E. M. McCreight: Priority Search Trees, Report CSL-81-5, XEROX Corp., 1982.

[NHS 81] J. Nievergelt, H. Hinterberger, K. C. Sevcik: The grid file: an adaptable, symmetric multi-key file structure, Report No. 46, Institut für Informatik, ETH Zürich, 1981.

[NiPr 82] J. Nievergelt, F. P. Preparata: Plane-Sweep Algorithms for Intersecting Geometric Figures, CACM 25, 10, 1982, 739 - 747.

[ShHo 76] M. I. Shamos, D. Hoey: Geometric intersection problems, 17th Annual Symposium on Foundations of Computer Science (IEEE), 1975, 208 - 215.


DATE
ILME